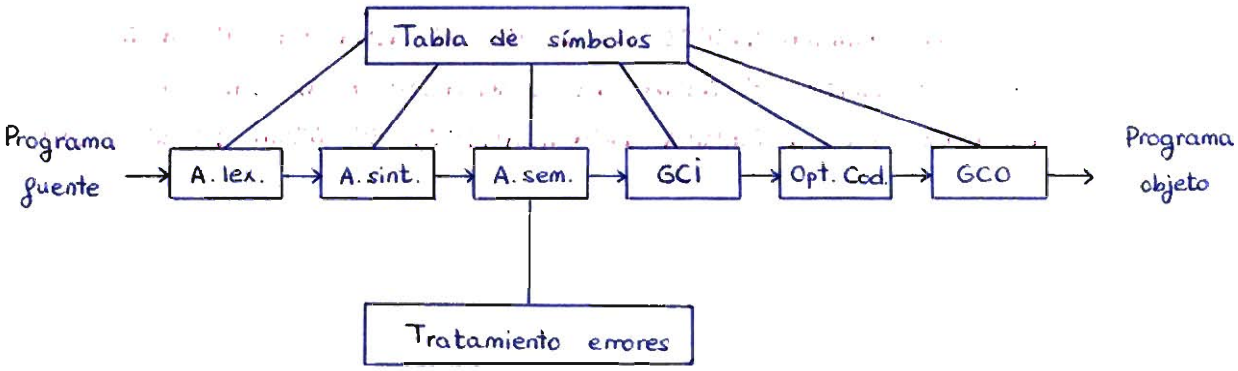
TEMA 3: TABLA DE SÍMBOLOS.

La fase de análisis está orientada al programa fuente. Está formada por el analizador léxico, el analizador sintáctico y el semántico.

La fase de síntesis está orientada al programa objeto. Está formada por el generador de código intermedio, el optimizador de código y el generador de código objeto.

El módulo de tratamiento de errores y la tabla de símbolos interactúan con todos los módulos.



La tabla de símbolos es una estructura de datos donde se va a almacenar información relativa a los símbolos (identificadores) que aparecen en el programa fuente.

¿Por qué es necesaria la TS? Porque la información relativa a los identificadores puede ser detectada (obtenida) durante la fase de análisis y es posible que otros módulos la necesiten posteriormente. No puedo ir pasando toda la información de un módulo a otro, generaría un caos. Por tanto, almaceno lo necesario en un lugar común con el que todos puedan interactuar.

La TS es exclusiva para los identificadores. Sólo existe en tiempo de compilación. Cuando ésta termina, la TS desaparece.

1. ESTRUCTURA DE LA TS.

¿Qué información hay que almacenar de los identificadores?

* Nombre (lexema) del identificador.
* Tipo (entero, bool, float, procedure, function, ...)
* Dimensiones (para matrices, vectores, cadena de caracteres, ...)

- Alcance o visibilidad del identificador (zona del programa en la que el símbolo es válido).
- Dirección de memoria asignada.
- Si es un procedimiento: número de argumentos, tipo de argumentos, modo de paso de los parámetros (por valor o por referencia) ...
- Si es una función: además de lo que necesita un procedimiento, debo tener el tipo de retorno.
- Valor correspondiente NO!!!

> EL VALOR CORRESPONDIENTE AL ID. NO SE ALMACENA EN LA TS.
> NO LO NECESITO EN TIEMPO DE COMPILACIÓN Y ADEMÁS LO DESCONOZCO: SE DETERMINA DINÁMICAMENTE EN TIEMPO DE EJECUCIÓN.

Ejemplo: Fragmento de programa en Pascal:

```
Var  a, aux : Integer;
Begin
     a := aux * cont;
End
```

T.S.	Atributos
a	
aux	

Como el identificador "cont" no se encuentra en la TS, se generará un error de identificador desconocido.

Para las palabras clave (palabras reservadas) tengo 2 posibles estrategias:

a) Incluirlas en la TS con un atributo que indique si es palabra reservada. Se realiza una generación automática de la TS con las palabras reservadas del lenguaje incluidas.

b) El analizador léxico tendrá un estado final para cada una de las palabras reservadas. Esto implica complicar mucho el a. léxico y tener un autómata gigante.

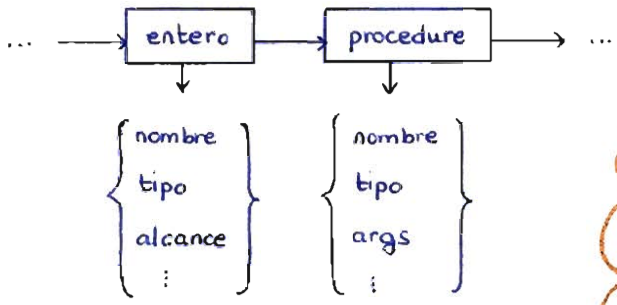
El analizador léxico simplemente introduce el lexema del identificador en la TS, pero no es misión del a. léxico recordar otra información o conocer el tipo del identificador. Esta distinción la tiene que hacer el analizador semántico y para ello incluimos la información de los atributos de los identificadores a través de las acciones semánticas. Después de encontrar un token, el a. léxico empieza de nuevo. No recuerda nada.

Para representar la TS se utilizan tablas o listas enlazadas. Las operaciones de la TS son las siguientes:

- Buscar identificador.
- Añadir identificador.
- Acceder información concreta de identificador.
- Añadir información del identificador.

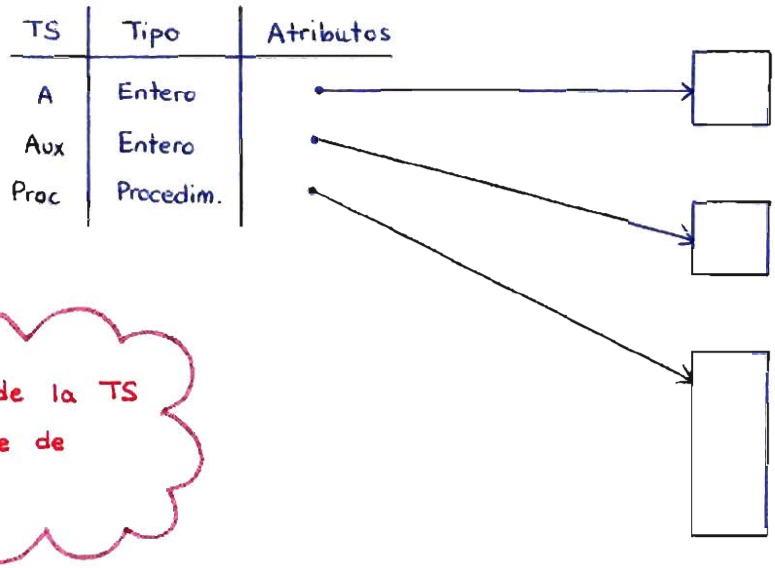
Al inicio el compilador crea una TS vacía o con las palabras reservadas del lenguaje.

No conviene usar una única tabla de estructura fija porque necesitaré almacenar unas atributos u otros dependiendo del tipo del identificador. Para no desaprovechar espacio con una tabla fija, se usará una lista enlazada en la que cada eslabón es un identificador.



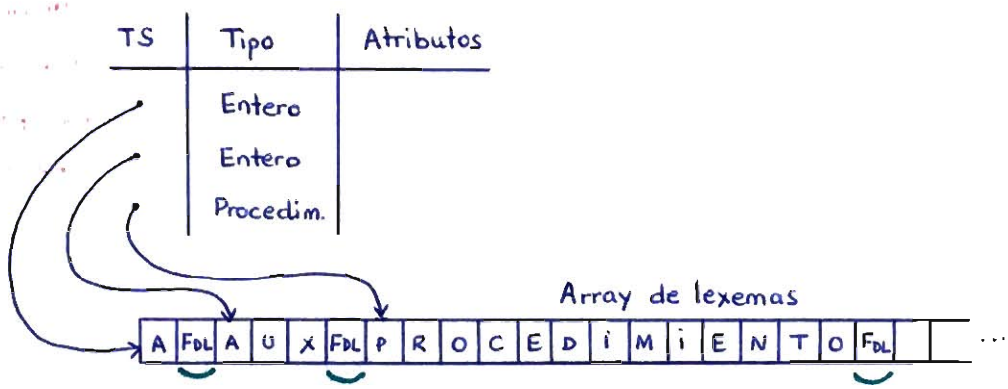
La información relativa a cada entrada se encuentra en una estructura dinámica adicional para no desperdiciar espacio.

Estructura de cada entrada de la TS:



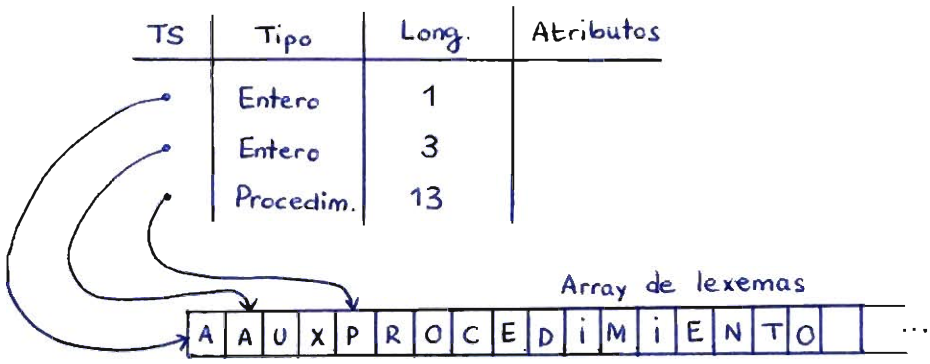
El tamaño de la TS no se conoce de antemano.

También podemos encontrar un problema con el lexema de los identificadores. Hay lenguajes en los que los identificadores tienen una dimensión reducida, pero en otros esta longitud es ilimitada o tiene un límite muy alto. Entonces, ¿qué se hace con el lexema? Si lo guardo en una zona fija tengo que reservar una zona de memoria muy grande, pero como la mayoría de los identificadores son de una longitud reducida (3-4 caracteres), esta solución nos llevaría a desaprovechar mucho espacio. Por tanto, el lexema se saca de la TS y se sustituye por un puntero a una zona de memoria donde se almacene el lexema.



FDL ≡ Caracter especial. Delimitador.

Alternativa al uso del caracter especial FDL ⇒ Para ver cuándo termina el lexema podemos incluir un nuevo campo en la TS que indique la longitud del lexema.




```
Program ...
  a
  :
  a
End
```

Al encontrar la primera "a", la inserta en la TS en, por ejemplo, la posición 37. Al encontrar la siguiente "a", el analizador léxico dirá que ha vuelto a encontrar el identificador de la posición 37.

2. ORGANIZACIÓN DE LA TS.

La tabla de símbolos se puede organizar de las siguientes formas:

1) TS LINEAL O CON FALTA DE ORGANIZACIÓN.

Cada vez que aparece un identificador nuevo lo introduzco en la primera posición libre de la TS.

La búsqueda de un identificador es secuencial, pudiendose empezar por el principio o por el final de la tabla. Es más lógico empezar por el final porque se tiende a utilizar las variables declaradas más recientemente.

Con este tipo de organización de la TS la búsqueda tiene un alto coste, pues es muy lenta. Insertar es muy fácil, pero buscar (que requiere recorrer toda la lista) se usa con mucha más frecuencia.

Ejemplo: Declaración: A, Aux, I, Cont, X.
Sentencias: I = Z.

TS	Lexema
1	A
2	Aux
3	I
4	Cont
5	X

Al ir leyendo las sentencias busco el identificador I en la TS y lo encuentro. Luego busco la Z y, al no encontrarla en la TS, se genera un error de identificador no encontrado.

2) TS ORDENADA.

Consiste en ordenar mediante algún criterio las entradas de la TS. Este criterio puede ser, por ejemplo, el orden alfabético.

Ahora insertar es más costoso, pero la búsqueda es más rápida que la secuencial porque se puede utilizar la búsqueda binaria (empezar a buscar por la mitad y seleccionar cada vez la mitad que me interesa).

$O(\log_2 N)$

Ejemplo: Declaraciones: A, Aux, I, Cont, X.
Sentencias: I = Z.

TS	Lexema
1	A
2	Aux
3	Cont
4	I
5	X

3) TS HASH.

Consiste en utilizar, además de la tabla, una función hash que, dado el identificador, localiza la posición de la tabla en la que debe ir.

La función hash podría ser, por ejemplo, el número de caracteres del lexema. Ésta es una función mala porque me da la misma posición para muchos identificadores.

El acceso a la TS mejora mucho si se utiliza una buena función hash.

↳ Colisión.

a) **Tabla hash abierta:** Con esta tabla, si la posición está ocupada, coloco el identificador en la primera posición libre más cercana a la posición original.

No es una buena solución, pues el número de colisiones es alto.

Ejemplo: Declaraciones: A, Aux, I, Cont, X.

Función hash: Nº de caracteres del lexema.

$f(A) = 1$

$f(Aux) = 3$

$f(I) = 1 \rightarrow$ Lo coloco en 2 porque 1 está ocupado.

$f(Cont) = 4$

$f(X) = 1 \rightarrow$ Lo coloco en 5 porque 1 está ocupado.

TS	Lexema
1	A
2	I
3	Aux
4	Cont
5	X

b) **Tabla hash con desbordamiento:** Funciona igual que la tabla hash abierta hasta que se produce una colisión. Ahora, cuando hay una colisión se lleva a una tabla de colisiones (tabla de desbordamiento).

Además, para facilitar la búsqueda, todas las variables con igual función hash se encadenan.

Ejemplo: Declaraciones: A, Aux, I, Cont, X, Pro, D.

Función hash: Nº de caracteres del lexema.

→

$f(A) = 1$

$f(Aux) = 3$

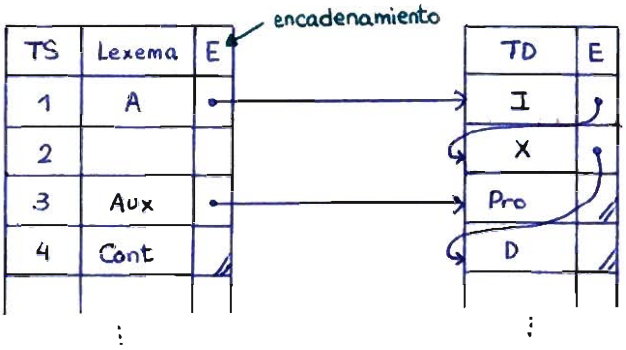
$f(I) = 1 \rightarrow$ Colisión => Lo llevo a la tabla de desbordamiento (TD).

$f(Cont) = 4$

$f(X) = 1 \rightarrow$ Colisión => Lo llevo a la tabla de desbordamiento (TD).

$f(Pro) = 3 \rightarrow$ Colisión => " " " " "

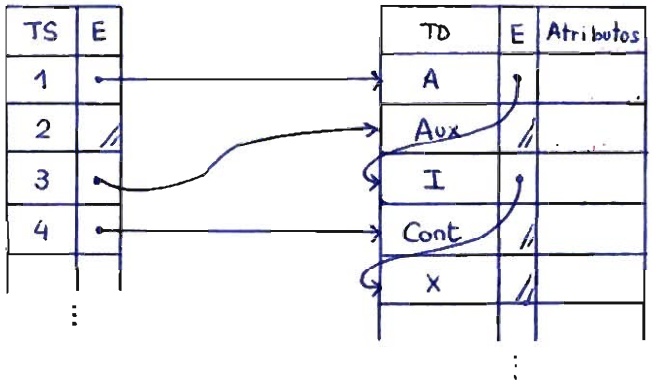
$f(D) = 1 \rightarrow$ Colisión => " " " " "



c) **Tabla hash con encadenamiento:** Funciona igual que la tabla hash con desbordamiento, pero no se almacena ningún identificador en la TS, sino que todos van a la TD.

Ejemplo: Declaraciones: A, Aux, I, Cont, X

Función hash: Nº de caracteres del lexema.



3. ALCANCE DE LOS SÍMBOLOS.

Es necesario definir las zonas de alcance (zonas de visibilidad) y la jerarquía de las zonas de un programa.

La mejor forma de representar el alcance de los símbolos en la TS es crear una TS individual para cada zona. Para representar la jerarquía se utiliza una tabla de bloques, que indica cuál es la tabla padre de cada TS individual.

Enlaza las TS individuales.

Además, el compilador necesita 2 variables globales (flags):

- **Modo declaración / Modo uso:** Indica si está en la zona de declaración de variables o en la zona de uso (zona de sentencias). Lo actualiza el a. sintáctico.
- **Tabla_actual:** Indica cuál es la tabla en la que se encuentra, en la que estoy insertando o buscando identificadores.

UNA TABLA DE BLOQUES + UNA TS POR ZONA.

Sólo hay una tabla de símbolos, pero internamente está dividida en bloques. Las tablas locales no contienen las palabras reservadas.

Así permitimos la implementación de procedimientos anidados.

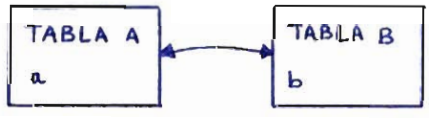
Existen 3 tipos de variables:

- **Variables globales:** Visibles en todo el programa fuente.
- **Variables locales:** Visibles únicamente en una determinada zona.
- **Variables no locales:** No es visible para todo el programa fuente, pero no es propia de alguna zona en la que sí es visible. Provoca zonas de alcance solapadas. Ocurre en el anidamiento de procedimientos.

Ejemplo:

```
Procedure A
  Var a;   ← Variable no local.

  Procedure B
    Var b;
    a = 8;
```



Al encontrar la variable "a", la busca primero en la TS de B. Al no encontrarla, la busca en la TS de A.

Ejemplo 1 (transparencia)

```
{Primer Ejemplo}
PROGRAM Ejemplo;
   VAR a, b: INTEGER;        Var. globales.

PROCEDURE proc1 (VAR x: INTEGER);
   VAR b: INTEGER;           Var. local.
BEGIN
   b:= 3;
   x:= b * 2 + a
END;        {fin de proc1}

PROCEDURE proc2 (d: INTEGER);
   VAR      u: INTEGER;      Var. no locales.
            v: BOOLEAN;
   FUNCTION fun (x: INTEGER): BOOLEAN;
            VAR a: INTEGER;  Var. local.
   BEGIN
            a:= 2;
            return (u MOD a) = b
   END;     {fin de fun}
BEGIN       {proc2}
   u:= a + d;
   v:= fun (u)
END;        {fin de proc2}

BEGIN       {principal}
   a:= 1;
   proc1 (a);
   proc2 (a);
   fun (b)
END.        {fin de principal}
```

Pascal es un lenguaje de bloques. Permite procedimientos anidados. Aparecen variables no locales.

Inicializar los flags:

- Def / Uso := Def;
- TS Actual := ∅;

BLOQUE TS	BLOQUE PADRE	PUNTERO TS
∅	—	
1	∅	
2	∅	
3	2	

TS ∅ PROGRAM	LEXEMA	TIPO	ATRIBUTOS ...
∅	Ejemplo	Program	
1	a	Integer	
2	b	Integer	
3	proc1	Procedure	1 arg: Integer
4	proc2	Procedure	1 arg: Integer

TS 1 PROC 1	LEXEMA	TIPO	ATRIBUTOS ...
∅	x	Integer	
1	b	Integer	

TS 3 FUN	LEXEMA	TIPO	ATRIB. ...
∅	x	Integer	
1	a	Integer	
2	fun	Boolean	

No lo puedo poner aquí.

TS 2 PROC 2	LEXEMA	TIPO	ATRIBUTOS ...
∅	d	Integer	
1	u	Integer	
2	v	Boolean	
3	fun	Function	1 arg: Integer Valor Boolean DirMem → Info. para poder usar "fun" como una variable.

Info. para poder usar "fun" como una función

ESTRUCTURA DE LA TABLA DE SÍMBOLOS

Notas Ejemplo 1:

1) Con el procedimiento (proc1 o proc2) tengo que crear una nueva zona de declaración de variables. Las variables definidas son internas al procedimiento y por ello no las puede usar el programa principal.

2) Con el campo BLOQUE PADRE (o bloque contenedor) de la TS indico cuál es el bloque padre del bloque de esa entrada. Este campo es necesario para ver el alcance de las variables.

3) Cuando llega el final de un procedimiento hay que volver a la tabla padre, borrar la TS correspondiente a ese procedimiento y eliminar de la tabla principal (tabla de bloques) la entrada correspondiente a ese bloque.

4) ¿Qué hago con la función? El identificador de la función se define en el bloque padre (TS 2) y se crea su tabla de símbolos (TS 3).

Cuando llego a la asignación fun:= (u mod a) = b, ¿cómo lo hago?

Para el identificador "fun" tengo dos posibles valores:

- Es función (recibe uno o varios argumentos).
- Es una variable (para poder guardar su valor de retorno).

Una forma de hacerlo es:

- Añadir en TS 2 (TS PROC2), es decir, la TS del bloque padre de la función, una entrada indicando que es una función y qué parámetros necesita.
- Añadir en TS 3 (TS FUN), es decir, la TS del bloque de la función, una entrada indicando que es una variable.

¿Cuál es el problema de esto? ¿Por qué no funciona? Porque Pascal permite funciones recursivas y esto no funciona con llamadas recursivas a la función.

Ej.- fact := n * fact(n-1);

"fact" es la función factorial.

Con este ejemplo, al llegar al primer "fact", el analizador léxico ve que es un identificador, que es correcto, lo busca en la TS correspondiente y ve que es una variable de tipo entero. Al continuar con el análisis y llegar al segundo "fact", lo busca y vuelve a encontrar que es una variable. Luego ve "(n-1)", por lo que da error de compilación. Esto ocurre porque el analizador léxico es muy tonto y es incapaz de distinguir si el identificador es una variable o una función.

Solución.- Tener una única entrada de "fun" (cualquier función) en la TS del bloque padre. En dicha entrada ponemos como información lo referente a la función (nombre, argumentos, modo paso de argumentos, tipo devuelto) y además la dirección de memoria correspondiente.

5) Con una TS de estas características estructurales es posible utilizar el mismo nombre de variables para zonas independientes de código (variables con el mismo nombre pero distinto alcance).

6) En el programa principal, cuando se lee "fun(b)", se busca en TS 0 (TS PROGRAM). No se obtiene información de "fun" porque ésta es una función local a "PROC2" y no es visible por el programa principal. Por tanto, hay que devolver un error de identificador no definido.

Ejemplo 2 (transparencia)

```
/*Segundo Ejemplo*/

int a, b;        Var. globales

void proc1 (int *x)
{
        int b;          Var. local.
        b = 3;
        *x = b * 2 + a;
}

boolean fun (int x)
{
        int a;          Var. local.
        a = 2;
        return (u % a) == b;
}

void proc2 (int d)
{
        int u;          Var. local.
        boolean v;
        u = a + d;
        v = fun (u);
}

void main ()
{
        a = 1;
        proc1 (&a);
        proc2 (&a);
        fun (b);
}
```

En C ∄ anidamiento de procedimientos.

↓

Sólo ∃ variables globales y locales a un procedimiento.

↓

No se necesita jerarquía de tablas.

↓

No se necesita tabla de bloques.

↓

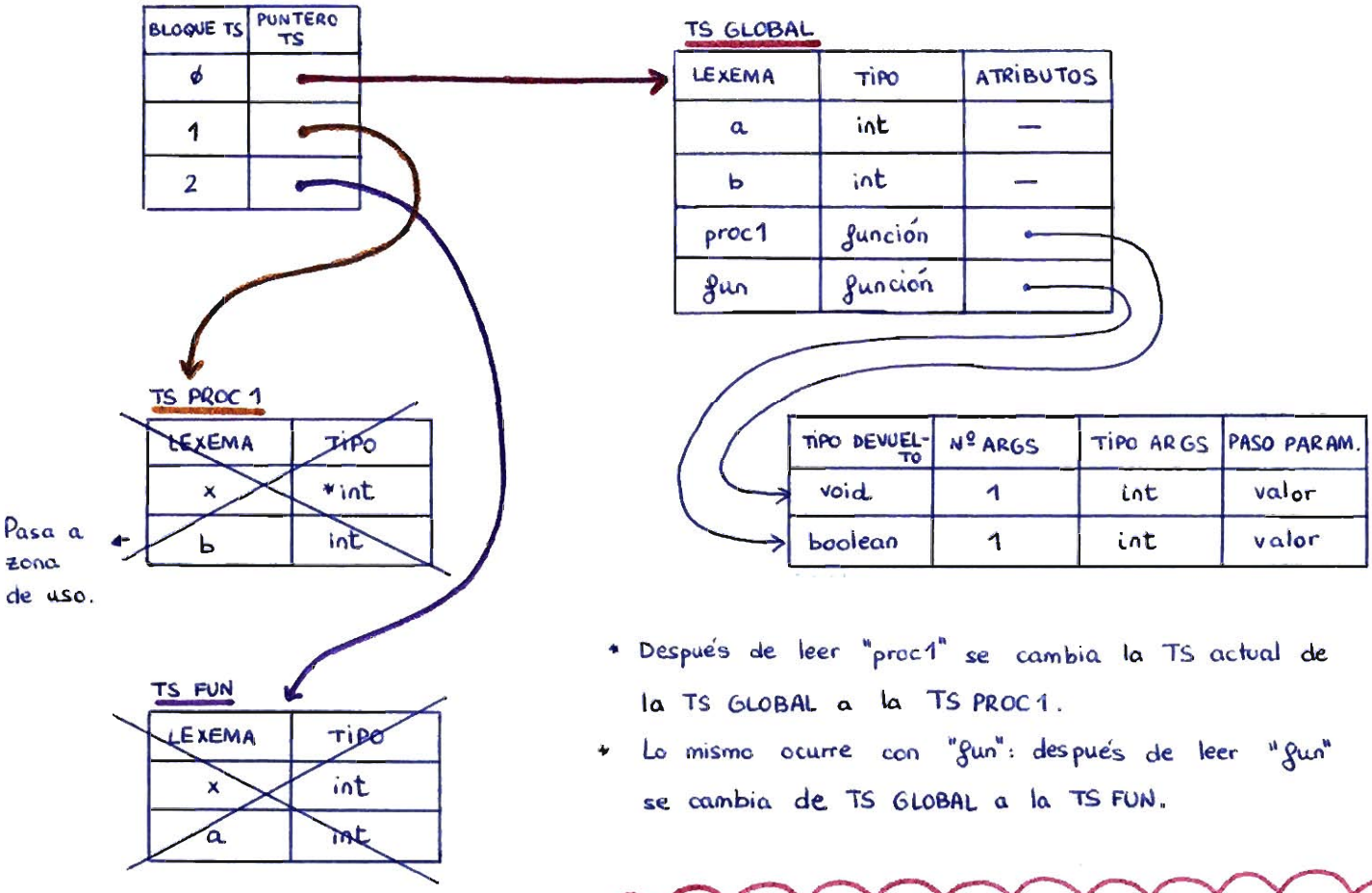
TS GLOBAL + TS POR BLOQUE

Inicializar los flags:

- Def / Uso := Def;
- TS Actual := TS Global;

} Los modifica (actualiza) el a. sintáctico.

ESTRUCTURA DE LA TABLA DE SÍMBOLOS.



- Después de leer "proc1" se cambia la TS actual de la TS GLOBAL a la TS PROC 1.
- Lo mismo ocurre con "fun": después de leer "fun" se cambia de TS GLOBAL a la TS FUN.

En C, para el MAIN no hace falta definir una TS nueva.

Notas Ejemplo 2:

1) En C, el nombre de la función nunca va a poder actuar como variable, por lo que es más sencillo que en Pascal. En C la función se comporta igual que el procedimiento.

2) No se pueden anidar los procedimientos. Como consecuencia, sólo hay variables globales a todos los procedimientos / funciones y variables locales a un procedimiento / función. Esto facilita el diseño de la TS ya que no hay ninguna jerarquía. Es por ello que no se necesita una tabla de bloques para conocer el bloque padre Simplemente tendremos la TS principal y una TS para cada procedimiento / función.

3) Al buscar una variable, busco en el bloque (tabla) actual y, si no aparece, la busco en la tabla principal. Si en esta tabla tampoco aparece se devuelve un error de identificador no definido. De hecho, en el ejemplo ocurre esto: cuando se alcanza "return (u % a) == b" el a. léxico da error: identificador "u" desconocido (no lo encuentra ni en TS FUN ni en TS GLOBAL).

4. TRATAMIENTO DE REGISTROS.

Tengo el siguiente registro:

```
struct A {          // Definido en C.
  int x;
  int y;
  ...
}
```

¿Cómo se inserta en la TS?

Cuando aparezca la declaración (definición) de un registro se añade en la TS correspondiente una entrada con el nombre del registro, indicando que es un registro e incluyendo un puntero a una TS adicional, que contiene una entrada por cada campo del registro.

Ejemplo:

```
A x;        // Declaración variable x del tipo A.
A z;        // Declaración variable z del tipo A.

x.x         // Acceso al campo x de la variable x.
u = z.x;    // Acceso al campo x de la variable z.
```

Lo más apropiado es poner en la TS global la dir. de comienzo del registro y el tamaño total. En la TSA se ponen las dirs. relativas de cada campo (el desplazamiento respecto al comienzo del registro).

TS GLOBAL

	LEXEMA	TIPO	CAMPOS
0	A	Registro	→
1	x	A	→
2	z	A	→
3	u	int	—

TS A

	LEXEMA	TIPO	...
0	x	int	...
1	y	int	...

TS x

	LEXEMA	TIPO	DIR. MEM.
0	x	int	...
1	y	int	...

TS z

	LEXEMA	TIPO	DIR. MEM.
0	x	int	...
1	y	int	...

Para generar el código objeto intermedio y final necesito conocer la dirección de memoria (el valor no) en la que almacenar el valor de cada campo. Por ello, en cada definición de variable definimos una TS adicional con cada uno de los campos y la dirección de memoria correspondiente.

¿Cómo son los tokens que genera el a. léxico?

Posibles accesos a registro:

```
x.x          // Acceso sencillo.
x[8].x       // Vector de registros.
z.x.a        // Registro cuyo campo es otro registro.
```

✗ Primera posibilidad:

```
< id, TSx(x) >     // Mirar en TSx en la dirección donde aparezca x.
< id, TS.8x(x)>
< id, TSz(TSx(a)) >
```

Complico los tokens y el a. léxico, que debe darse cuenta de si está accediendo a un registro o a algo más complicado. Para reconocer los accesos concretos el a. léxico necesitaría un pequeño a. sintáchico.

✓ Segunda posibilidad:

```
< id, TS(x) >, < ., punto >, < id, TS(x) >
< id, TS(x) >, < [, corch_ab> , ... , <., punto>, < id. TSx (x)>
```

El a. léxico genera los tokens para cada símbolo. Los accesos se realizan token por token. Será el a. semántico el que se dé cuenta de que es un acceso a registro e irá cambiando a la TS que corresponda.

Ejemplo: vector [7] of A semana;

TS GLOBAL

LEXEMA	TIPO	CAMPOS	SUBTIPO	DIR. MEM.	TAMAÑO	DIM.	
Semana	Vector		A			7	0 1 2 3 4 5 6

TS_0	Tipo	Dir.
x	int	...
y	int	...

Ejemplo: Fragmento de código:

```
struct z {
   A x;          // Campo x de tipo A: El campo x es, a su vez, otro registro.
   ...
}
struct A {
   int a;
   ...
}
z.x.a;
```

¿Qué tokens se van generando?

Es la sentencia la que genera tokens. Los "struct" no.

z → < id, TS (z) > → Tabla actual = TS.

. → < . , Punto > → Tabla actual = TS_z (El a. semántico se da cuenta de que es un registro).

x → < id, TS_z(x) > → Tabla actual = TS_z

. → < . , Punto > → Tabla actual = TS_{zx} (El a. semántico vuelve a ver que es un registro).

a → < id, TS_{zx}(a) → Tabla actual = TS_{zx}

Al comunicar el token "punto", mediante acciones semánticas hay que indicar la tabla en la que está el campo y cambiar a ella.

```
{Primer Ejemplo}
PROGRAM Ejemplo;
          VAR a, b: INTEGER; {globales}

PROCEDURE proc1 (VAR x: INTEGER);
          VAR b: INTEGER;
BEGIN
          b:= 3;
          x:= b * 2 + a
END;      {fin de proc1}

PROCEDURE proc2 (d: INTEGER);
          VAR       u: INTEGER;
                    v: BOOLEAN;
          FUNCTION fun (x: INTEGER): BOOLEAN;
                    VAR a: INTEGER;
          BEGIN
                    a:= 2;
                    return (u MOD a) = b
          END;      {fin de fun}
BEGIN     {proc2}
          u:= a + d;
          v:= fun (u)
END;      {fin de proc2}

BEGIN     {principal}
          a:= 1;
          proc1 (a);
          proc2 (a);
          fun (b)
END.      {fin de principal}


/*Segundo Ejemplo*/

int a, b;           /*globales*/

void proc1 (int *x)
{
          int b;
          b = 3;
          *x = b * 2 + a;
}

boolean fun (int x)
{
          int a;
          a = 2;
          return (u % a) == b;
}

void proc2 (int d)
{
          int u;
          boolean v;
          u = a + d;
          v = fun (u);
}

void main ()
{
          a = 1;
          proc1 (&a);
          proc2 (&a);
          fun (b);
}
```